Title: MCNP 2020 - White Paper - Preparing LANL Monte Carlo for Exascale Computer Systems

Author(s): Brown, Forrest B.
Kiedrowski, Brian Christopher
Bull, Jeffrey S.
Cox, Lawrence James

Intended for: MCNP documentation & planning

Issued: 2015-04-08

MCNP 2020

- White Paper -

Preparing LANL Monte Carlo for Exascale Computer Systems

Forrest Brown,

Brian Kiedrowski, Jeffrey Bull, Larry Cox

Monte Carlo Codes Group (XCP-3) & XCP-DO

X Computational Physics Division

Los Alamos National Laboratory

July 22, 2013

--- blank page ---

Project:	**MCNP 2020**
Group:	XCP-3
Lead:	Forrest Brown, R&D Scientist 5
	Brian Kiedrowski, Jeffrey Bull, Larry Cox
Sponsors:	DOE-ASC, DOE-NCSP

Problem Statement:

The MCNP Monte Carlo code for simulating particle transport is the product of over 60 years of Monte Carlo code development at LANL. MCNP is the most-used code worldwide for particle transport in nuclear systems, with over 10,000 users in dozens of application areas. At LANL, MCNP is used extensively in applications for nuclear criticality safety, stockpile stewardship, homeland security, radiological safety, experiment design and analysis, etc., and is considered a critical technology resource for national defense.

MCNP is often referred to as the “gold standard” for radiation transport calculations, based on the long development history and extensive validation work. Today, however, this critical technology and its well-deserved reputation are at risk:

- Leading-edge HPC systems change radically over 5-10 year periods, with new designs, hardware, and system architectures. The past decade has largely been clusters of homogeneous conventional compute nodes; the next decade will involve emergent new architectures with massive numbers of nodes and processors, and potentially heterogeneous architectures. Software must evolve to match the new hardware; static software will be rapidly left behind.
- The core portions of the MCNP software are largely based on computing practices from the 1950-60s. Over the years, MCNP was occasionally updated to newer Fortran dialects, without major code restructuring. In the early 2000s, MCNP was completely converted to Fortran-90 syntax (without revising code structure or algorithms), and was enhanced to handle both MPI message-passing and OpenMP threading for parallelism. No major restructuring of the core portions of the code has ever been done. As a result, MCNP cannot make use of state-of-the-art processors such as vector GPUs, and could potentially use many-integrated-core processors (MICs) only by treating them as a mini-node in a cluster.
- With the recent merger of MCNP5 and MCNPX into MCNP6, code size has grown from 100K lines of code to 500K lines. Software complexity scales as some power of the code size, so that code maintainability now suffers. The time and effort required for even minor code improvements, for routine maintenance, and for testing has dramatically increased.
- MCNP now suffers from a malaise of “feature bloat” and “technical debt”. Many hundreds of features – code options for different user applications – have been added to the original code base, while very little effort has gone into overhauling the core portions of the code. A side effect of this expansion is that the current code runs 30-50% slower than it did a decade ago. The extensive feature set is required to support user needs, and cannot be eliminated or cut back. However, MCNP suffers from the lack of attention to the core software and infrastructure needed to support those features.

Proposed Solution:

We propose a targeted effort over the next several years to upgrade the core software portions of the MCNP code. This effort will focus on fundamental structural changes to the underlying MCNP algorithms, data storage, and code architecture. These changes will directly improve code maintainability, reduce the development time for new features, improve the code performance, and make the code more flexible and adaptable for expected future computer systems. To move forward with new computer systems and additional new features, it is vital to have a solid base.

This effort requires 3-4 FTEs each year on an ongoing basis, so that a well-coordinated holistic approach can be followed. The extensive changes required to the code infrastructure will be an evolution – neither a massive rewrite, nor a set of piecemeal revisions. This evolution will involve a small set of core developers, guided by extensive experience from successful past code restructuring efforts, and by extensive understanding of expected future computer systems.

In effect, this proposal suggests that a small team of 3-4 developers work closely together and focus on improving the structural foundation of MCNP, while other MCNP developers continue to focus on adding new features and analysis capabilities. These efforts are synergistic – a strong core enables future development; separating core foundations from added features leads to a coherent, better performing code foundation.

Comment - Cost:

An expected response to this proposal is that 3-4 FTEs per year for several years is too expensive to consider. Our response is: (1) A conservative estimate of the government funding at LANL for Monte Carlo codes (leading up to MCNP) over the past 60 years exceeds $300M. Such a huge investment cannot be allowed to atrophy. (2) This effort is an investment in the future of MCNP, to ensure that it can keep up with rapid computer system changes. (3) This effort makes up for accumulated past technical debt. A significant portion of this effort will be repaid by reduced costs for code development and maintenance, and by improved code performance. (4) We can't afford to not do this.

Comment - Competition:

30 years ago, MCNP was widely adopted worldwide for radiation transport calculations; there were few readily available alternatives. Today there are very many high-quality Monte Carlo codes widely available: Geant4 for high-energy physics, detectors, medical physics; MC21, MVP, Serpent, and OpenMC for reactor physics calculations; Scale/Keno for criticality safety; Penelope, EGSnrc, and ITS for electron transport. (Note that at LANL, Geant4 is the code of choice for muon detector research.) Those newer codes have a more modern code base, support fewer legacy features, and have more flexibility for being adapted to new computer architectures than MCNP.

Comment – Exascale Systems:

Exascale systems are expected to arrive within the next 5-10 years. They will likely have between 100,000 and 1,000,000 nodes and be able to execute up to a billion threads at any given time. Experience with leading-edge supercomputing systems over the past 30 years suggests that a 1000-fold increase in peak computer hardware performance leads to a 100-fold increase in performance of algorithmic kernels, and a 10-fold increase in the delivered performance of large, complex production codes for applications. Such 10-fold increases in computing capabilities, however, are the enablers for significant step changes in design capabilities and workflow for end-users.

Supercomputer makers will have to construct their machines to balance the cost, power consumption, processor performance, and interconnect speeds. An exascale machine should cost less than $200 million and use only about 20 MW, or about 50 Gflops per watt. Today's most energy-efficient processors are GPUs (Nvidia K20) and MICs (Intel Xeon Phi). Individual node performance should be 1-20 Tflops, and interconnects will need to have throughputs of 200-400 GB/sec. Heterogeneous combinations of general purpose CPUs, GPUs, and MICs are likely, a radical departure from the homogeneous commodity clusters of the last 10 years. While memory capacities will grow and faster SSD storage will become available, it is a near certainty that the amount of memory per computing core will decrease. There is also a strong possibility of in-memory computing, where some compute functions are performed directly by the memory chips to avoid data transfer costs.

From the standpoint of MCNP and its core software base, consider that it will have to adapt to: a 1000-fold increase in parallelism, reduced memory per core (hence extensive sharing), heterogeneous processing, and unknown connectivity. It will be necessary to eliminate synchronization, minimize communications, subdivide and dispatch work to different types of processors, perhaps implement some functional parallelism, and introduce some additional hierarchies in parallel algorithms. All of this work will require extensive testing, experience, and expert code development, and will take years to accomplish. It cannot be carried out at all, however, without a very solid, modernized, and enhanced core software base. The effort needs to start now.

Some Additional Discussion & Details:

For the modernization and refactoring of a large scientific code package, it is helpful to consider several general areas: code structure, data structures, algorithms, computational physics kernels, and infrastructure. Any modernization effort that attempts to change 2 or more of these areas simultaneously is doomed to certain failure. That is, when upgrading data structures, all other aspects of the code must be kept invariant; while upgrading algorithms, the data structures, etc., must not be changed.

At a high level, some general objectives of the modernization effort are:

- The overall code structure should reflect the algorithms that are being used, with explicit representation of high-level iterative loops (e.g., for timesteps, alpha cycles, k cycles, histories). The fundamental, indivisible kernel should be an individual particle history.

- A major, forward-looking goal of modernization should be to permit 1000s of threads per node, and many 1000s of nodes. There may be a need for some functional parallelism, e.g., with some nodes dedicated to global tallies, some to data retrieval, and most to compute-intensive processing of particle histories. The present master-node-thread parallel structure may need additional hierarchical elements, with "swarms" of particle histories dispatched to collections of nodes.
- Routines related to a particle history must be encapsulated, with explicit input and output arguments. There should be no file I/O, memory reallocation, synchronization, or interrupts within a single history. For the present, a single history should not send or receive MPI messages. For the future, single histories may need to invoke one-sided MPI remote memory access for retrieving data or making tallies. A particle history is the basic, indivisible unit for parallelism.
- Significant effort should be applied to reducing the memory required for a history. The past practice of replicating data and tally arrays for each history (to support parallelism without thread locks) should be revised. Threading critical sections, thread-buffered list tallies, and other schemes should be used to reduce memory usage while maintaining correctness.
- MPI message passing must be lean, and unnecessary data interchange should be eliminated. Explicit synchronization must be eliminated, and messaging and work allocation should be as asynchronous as possible. When rendezvous are required algorithmically, only necessary data should be collected, so that lightweight and heavyweight rendezvous operations are implemented.
- Fixed-source Monte Carlo calculations should be restructured to run batches of particle histories, much like criticality calculations, to reduce overhead and provide more logical control for the calculations.
- The current practice of using distinct, large computational routines to treat the histories of each type of particle should be revised (if possible) by the use of a common controller with lists of function pointers to handle particle-specific options.
- For large parallel computations, domain decomposition based on spatial regions should not be used. Instead, particle histories should continue to be the fundamental, indivisible parallel unit, with one-side MPI remote memory access operations for dealing with massive data and tally sets distributed over server nodes.
- To the extent possible, high-level code features should interface with the core transport, data, and tally routines, rather than requiring low-level modifications to the core routines.
- All side effects from function calls should be eliminated wherever possible. The 1960s-era practice of passing arguments in/out of routines via Fortran common blocks must be eliminated, and all arguments that may change should be passed explicitly in the calling interface.
- The names of various arrays and datasets should be revised to be more descriptive of their content and function. The 1960s-era practice of cryptic 4-6 character names should be eliminated.

The overall modernization effort must be carefully coordinated, for example changing some data structures, then some algorithms, then additional data structures, etc. Many incremental changes are required in sequence to achieve overall goals. A number of these important incremental changes are listed briefly below. The list is neither ordered nor complete, but is representative of the many incremental changes that are needed.

- Data structures: Reorganize some of the fundamental data arrays into a modern, structured form. Easiest initial targets include cell, surface, and some geometry information.
- Data structures: Reorganize data storage for tally arrays and statistical quantities into a modern, structured form. Current tally storage is obtuse and archaic, combining tally specifications and data of multiple different types and purposes into single flat arrays.
- Algorithms: Fix the inverted global logic (infinite history loop with conditional exits), to definite well-structured explicit loops with control logic.
- Infrastructure: Replace the current "runtape" sequential dumpfile by a structured direct-access file, with datasets accessible by name.
- Algorithms: Examine parallel threading, to permit 1000s of threads per MPI compute node.
- Algorithms: Eliminate the replication of many large arrays for parallel threads.
- Algorithms: As a step toward remote tally servers, use list tallying during particle random walks, combining list tallies afterwards. Also reduces threading critical section overhead. Also provides basis for future one-sided MPI for remote tallying or data fetches.
- Code structure: Eliminate side effects from subroutine and function calls (explicit argument passing for result variables, rather than hidden by common block updates)
- Algorithms: Make use of new sparse tally features for all large mesh tallies.
- Algorithms: Include random number generator seed information with particle history structures, not global code routines.
- Algorithms: Eliminate the use of parallel locking for source particle generation, including surface-source reading.
- Algorithms: Eliminate thread-private copies of tally arrays, by use of either shared tallies with threaded critical sections or by the use of list tallies.
- Code structure: Reorganize into subdirectories for base and features.
- Infrastructure: Upgrade the build system to be modular, building code components independently.
- Infrastructure: Upgrade the build system to provide distinct different target directories for MPI builds, specific compiler builds, etc.
- Infrastructure: Upgrade the build system to permit different compiler options for some routines, where required for optimization or correctness.
- Algorithms: Separate the parallel rendezvous into lightweight & heavyweight rendezvous, to eliminate many unneeded MPI messages.
- Algorithms: Use MPI collectives (reductions) for improved parallel accumulation of tally information.
- Algorithms: Create a particle stack, and later possibly event queueing.
- Data structures: Structured format for cross-section data.
- Algorithms: Collect information to estimate histories/sec, collisions/history, flights/history,

tallies/history, etc., to guide parallel implementation. Provides basis for adaptive parallelism.

- Infrastructure: Investigate parallel I/O with MPI for reduced dumpfile times. Needs to consider restarts with different numbers of MPI processes.
- Infrastructure: Simplify & unify the various configuration files for compilers, parallel, options, etc.
- Data structures: Reorganize modules and data residence to reflect code logic.
- Computational kernels: Untangle and simplify code logic for geometry tracking.
- Computational kernels: Make use of modern language features and standard functions to replace obsolescent or obscure coding.
- Code structure: Reduce complexity of input processing, and make use of parallelism for setting up very large problems.
- Infrastructure: Incorporate modular features into build system, for customer-specific builds.
- Algorithms: Eliminate naïve sorting algorithms, replace with either modern efficient sorting or hash-based algorithms.